From visual to textual programming

Easing the transition for children

Department of Computing, Faculty of Technology

Master of Science Thesis

Author:
Jennica Harju

Supervisor:
Jouni Smed

June 2025

Master of Science Thesis
Department of Computing, Faculty of Technology
University of Turku

Subject: Programming education
Author: Jennica Harju
Title: From visual to textual programming: Easing the transition for children
Number of pages: 51 pages
Date: June 2025

Children start learning how to program at young ages and visual programming is a great way to introduce them to the world of coding. However, at some point they will have to take the big leap from visual programming to textual programming if they want to deepen their programming knowledge and create programs that work outside of visual programming environments.

For some children this transition can be difficult. This thesis explores one way of making the transition easier by introducing a step between visual and textual programming. Visual Python is a visual programming tool where the user uses blocks to create code, but the blocks have the syntax of Python. This way children can familiarise themselves with the commands and new words before they have to type out the code themselves.

Visual Python was tested out by several groups of children who learn programming as a hobby. These children as well as their instructors answered a survey after completing one programming lesson with Visual Python. The results show that Visual Python is a beneficial tool for the transition. Although further research is needed and other approaches should be considered as well.

Keywords: teaching programming, visual programming, textual programming, scratch, python

Table of contents

1 Introduction

Programming is a valuable skill for children to learn at a young age, as it helps them understand the digital world around them better. Programming skills also allow them to shape this digital world if they wish to do so.

Children usually start learning how to program with visual programming tools that use drag-and-drop blocks such as Scratch. However, moving from visual programming to text-based programming languages like Python, can sometimes be challenging. This thesis explores if a visual programming tool that uses the syntax of Python, but is still block based, could help make this transition easier for children.

1.1 Motivation

Smart devices, digital media and online networks have become an increasingly important part of our everyday lives. Different devices are used for various purposes and people of all ages depend on technology for everyday tasks. This makes it an essential part of modern life. Technology will only continue to evolve and its role in our daily life will only become more significant.

Studies have shown that most children are using their smartphones daily. Children are also going online at younger ages nowadays. They use digital platforms for all kinds of activities such as learning, playing and talking to their friends and family. More and more of schoolwork is also done online and children are taught digital literacy at schools. However, for some children, smart devices can still seem like magical devices that work as portals to another reality. They might not fully understand how these devices operate or the implications of their use. [15, 18]

One of the main arguments for why children should learn programming, is that they need to learn how these smart devices function. By learning programming, they can better understand how the world around them works. Understanding the basics of programming not only demystifies technology, but also provides a solid foundation for its responsible use. By understanding these concepts, children can better navigate and understand the digital world. [2]

Programming also teaches children other valuable skills such as computational thinking: breaking problems into smaller more manageable parts, identifying similar patterns within problems, focusing on the important details as well as creating step-by-step solutions to solve a problem. Computational thinking is a valuable skill for everyone and it is useful beyond just technology-related fields. Despite the fact that programming focuses on the logical side a lot, it also nurtures creativity. Problem solving often requires creative solutions and thinking outside of the box. Creativity is also needed, for example, for game design and web development. [2]

Learning programming at a young age can also lead to better job opportunities in later life and not just in the field of information technology. In Europe, more than 90% of professional roles required at least a basic level of digital knowledge in 2023 [11]. Computational thinking, creativity and critical thinking skills are also valued more in the modern job market where the use of artificial intelligence is becoming more common [16].

These are some of the many reasons why more and more children are learning programming these days. Programming and computational thinking were added to the Finnish national core curriculum that came into effect in 2016. However, despite this inclusion, the implementation across schools have not been consistent. Just over half of Finnish upper secondary schools actually offered programming education in 2023. [13]

Primary schools have integrated programming into various school subjects such as mathematics and handicraft, but many of them struggle to make programming a cohesive part of their curriculum. The subject can feel disconnected for students and lacks continuity. A study published by the Finnish Ministry of Education and Culture was conducted between 2017 and 2019. It found that 89% of 9th-grade students in Finland were unable to score any points in basic programming tasks. The tasks tested logical reasoning, following instructions, and solving simple arithmetic problems. [19]

This is why parents want their children to learn programming outside of school environments as well. Many children are also excited to take on this new hobby as they might be interested in video games or have other technology related interests. Companies such as Suomen Tiedekoulu offer options for these children to learn programming after school. Tiedekoulu offers programming lessons at several different levels. Starting from the age children learn how to read, all the way up to adulthood.

1.2 Problem description

Teaching programming to children often starts with visual programming tools such as Scratch. These visual tools make it easy for beginners to learn the basic concepts of programming before moving on to more difficult text-based programming languages. Visual programming tools often use simple drag-and-drop blocks that connect to each other intuitively. The blocks are most often colour coded, intuitive to use and, if possible, in the user's own native language. This way, the children can focus on understanding the basic logic behind programming without having to worry about the stricter rules of text-based programming. Visual programming can also be more entertaining and motivating for children than textual programming. Visual programming is a common starting point for adult learners as well.

The next step in learning how to program is usually to learn a text-based language such as Python. Python has a relatively simple syntax compared to some other common programming languages. This is why it is a popular choice for beginners and is often the first text-based programming language at schools and other coding courses. However, the shift from visual programming to text-based programming can be difficult for some children. They need to pay close attention to writing the code correctly and follow the syntax rules. Especially younger children might not be used to using a keyboard and writing special characters such as brackets and colons. These small details are important when programming, but for beginners, even one misplaced comma can lead to much frustration when the program does not work as intended. Additionally, if the child is not a native English speaker, simple words like "and", "or", "if" and "else" might be completely new to them. This transition can be overwhelming for children even if they already understand, for example, the logic behind loops and conditional statements.

Suomen Tiedekoulu organises online and face-to-face teaching of robotics, programming and natural sciences for children of all ages. In their programming lessons, visual programming sites Scratch and MakeCode are used as the first steps to learn how to program. After learning the basic concepts of programming with these visual programming tools, the children start learning Python. However, after working at Suomen Tiedekoulu for over two years, I observed that this step from visual programming to textual programming can be a very difficult step for some children and they lose their motivation and some even quit their hobby entirely.

This thesis focuses on adding a step between visual and textual programming, where the user can program visually while also familiarising themselves with the syntax of Python. Visual Python was created for this purpose and it was tested by several coding groups at Suomen Tiedekoulu.

1.3 Research questions

The object of this thesis is to explore how to make it easier for children to transition from visual to text-based programming. To achieve this, several key questions must be addressed.

RQ1: Do children find it difficult to move from visual programming, using drag-and-drop blocks, to typing out code in languages like Python?

RQ2: Can the transition be made easier by breaking it down and adding a step between visual and textual programming? Showing Python in a block format before moving fully to text might make it easier for children to understand Python syntax before having to actually write it themselves. Does this method actually help children learn Python better before they start writing it themselves?

RQ3: If children use a visual version of Python, will it help them understand the actual text-based Python code better before they ever type it themselves?

These research questions aim to find if there is a need for an additional step between visual and textual programming and can Visual Python be a solution for it.

1.4 Review of current research

Several studies have been conducted that show how visual programming is useful for learning the basics of programming and improving computational thinking especially among children [2, 12, 16]. However, research done on the topic of transitioning from visual programming to textual programming is more parse.

Armoni et al. [1] study whether learning Scratch in middle school can lead to improved learning of computer science at the secondary level and how it contributes to the improvement. Their results provide strong evidence to justify learning visual programming at

a younger age before transitioning to textual programming. They mention several challenges the students had when transitioning to textual programming.

One major difficulty was adapting to the English keywords in C# and Java, as Scratch supports multiple natural languages and the students are used to programming in their own language. Students also struggled with recognising familiar concepts in new forms, such as variables and loops, due to the differences in implementation. However, despite their initial confusion, they quickly adjusted to typed variables and bounded loops, showing that these challenges were temporary. [1]

Tóth and Lovászová [19] noticed a significant gap between students' ability to program in an educational visual environment and their ability to write code in a professional programming environment. Their area of interest is mobile application programming and they focus on the transition between MIT App Inventor 2 and Android Studio. To help with this transition, they propose the Java Bridge tool, which includes a library and a code generator. The library maintains App Inventor's terminology in Android Studio, while the generator converts visual projects into Java code. Their method follows three phases: starting with visual programming, moving to hybrid coding with Java Bridge, and finally transitioning to full text-based programming. Adding a hybrid solution as a step between visual and textual programming is similar to the concept of Visual Python with the additional benefit of being able to generate text-based code from visual code.

Cheung et al. [3] found that in grades 11 to 13 students often find visual programming too limited and textual programming too difficult. They propose a text-enhanced graphical programming environment called BrickLayer. On BrickLayer the user can build programs using drag-and-drop blocks and also see the corresponding Python syntax in real-time. The environment also lets them modify the code freely. They had positive results at workshops and found that nearly 60% of the students were motivated to edit the textual source code by themselves.

Kaurel [7] made the same observation about the difficulty of transitioning from visual to textual programming. He noted the problem while working at a Norwegian coding club. Many participants dropped out of the Python courses because they struggled to understand the concepts of Python. To solve this problem, Kaurel explores the idea of adding two steps between visual and textual programming. Between Scratch and Python, he introduces a new language, Klang. Klang would first be programmed visually and then textually. However, the

interface of visual Klang was not polished and the children criticised it highly. They also did not get to test it out themselves. The evaluation was done with interviews with focus groups. His results are preliminary, but show that having an environment, where visual and textual programming could be used interchangeably, can be a viable solution for the problem. One language programmed visually and textually was easy for the children to understand. The children could picture using the visual syntax as a fallback mechanism when they did not remember some part of the textual notation.

This same concept is also shared with Visual Python. One language written first visually and then textually. However, with Visual Python, there is no need for children to learn another new language. On the other hand, Klang has a more intuitive syntax than Python and could still be beneficial for children to learn first.

1.5 Structure of work

This thesis focuses on the transition from visual programming to textual programming and how an extra step between the two could help ease the transition.

The thesis consists of five chapters. The first chapter is the introduction. It starts by presenting the motivation for choosing the topic and why it is important. After the motivation, the problem is described in more detail and the research questions are defined. The chapter also includes an overview of other research related to the topic.

The second chapter focuses on background information that is important for understanding the topic. It starts by explaining what programming languages are in general and what syntax and semantics mean in this context. Then visual programming is explained in more detail and visual programming tools Scratch, MakeCode and Blockly are examined more closely. The focus is on the syntax and semantics and what is unique about these programming languages. Finally, a textual programming language called Python is introduced. Just like with visual programming, the focus is on the syntax and semantics of this language.

The third chapter explains how Visual Python was created to ease the transition from Scratch and MakeCode to Python. The chapter also explains how Visual Python works.

The fourth chapter focuses on the evaluation of Visual Python. The chapter starts by introducing the children who tested Visual Python. The lesson structure and the exercises

used for the testing are also described in this chapter. After that, the results of the survey that the children answered after completing the exercises are presented. The instructors, who instructed these lessons, also answered a survey afterwards and the results are presented at the end of this chapter.

The fifth and final chapter is the conclusion. It discusses and answers all the research questions that were presented in the first chapter. Future work is also discussed at the end of the thesis.

2 Background

This thesis focuses on transitioning from visual to textual programming. This is why it is important to understand how programming languages work and what are the main differences between them, especially with their syntax and sematic. This chapter begins by explaining what programming languages are and what syntax and semantics mean in this context. Then, visual programming tools that are related to this thesis are explained. Finally, the chapter provides an overview of Python with the focus remaining on its semantics and syntax.

2.1 Programming languages

Programming is the process of writing a set of instructions for a computer to follow and execute. These instructions are called code and they can look very different depending on which programming language is used. Just like how people communicate with each other in different languages, we use different languages to communicate with computers. Nowadays, programming is usually done with high-level languages such as Python, Java and C#. These languages are easier for the programmer to write and understand. Because computers don't understand these high-level-languages directly, they are then converted to low-level languages, which consists of binary code or machine instructions that are in a format that the computers can read. [6]

The main difference between natural languages and programming languages, is that programming languages need to have only one way to interpret them. There is no room for ambiguity as computers require clear and precise instructions to follow them correctly. While humans can understand things like context, tone and implied meaning, computers are exact. They follow commands literally and this means that every detail must be correct. Even small errors, like a missing semicolon or a wrong character, can make the program behave unexpectedly or not work at all. [6]

2.1.1 Syntax

Syntax refers to the words and symbols of a language and how to write them down in a meaningful way. It can be described as the grammar of a programming language. Just like

grammar rules how we should structure sentences in natural languages, syntax rules how a code should be written so that the computer can understand it. [6, 8]

Different programming languages can do the same thing while looking different. For example, here is how to write a simple loop that prints the text “Hello world” three times in a programming language called Java:

```
for (int i = 0; i < 3; i++) {
      System.out.println("Hello world");
}
```

And here the same code is written with Python:

```
for i in range(3):
      print("Hello world")
```

While both of these codes do the same thing, the user must be familiar with the syntax of the programming language to be able to write and understand them. For example, Java uses curly brackets to enclose code blocks while Python uses indentations instead. Python has a relatively simple syntax and writing a simple command such as print is much more intuitive for new users. However, the difference in syntax between visual programming and text-based programming is much larger than the differences between different text-based programming languages.

2.1.2 Semantics

While syntax rules how a program looks, semantics defines how it functions. Semantics is about understanding the meaning behind what is written and knowing what the program does when it is executed. In other words, while syntax defines the structure, semantics defines the behaviour. [6, 8]

We can also compare this to natural language. The sentence “Raining now outside” is grammatically wrong so the syntax is incorrect. However, the meaning of the sentence is still clear. Even with the wrong syntax, the sentence can still be understood. In everyday conversation, we often rely on context to figure out what someone is saying, even if they do not use perfect grammar or full sentences. Unfortunately, this flexibility does not exist in programming and the instructions given to the computer need to be exact. [6]

In programming, if the syntax is correct, but the semantics is wrong, then the program will not work as expected. Even if it is written correctly, the logic behind it must be flawless. If the semantics is correct, but the syntax is wrong the program might not work at all or do something not wanted. Even the tiniest of errors, can cause this to happen. This is why both syntax and semantics are important when programming and the programmer must be familiar with both of them. [6]

All programming languages share similar semantics. Common programming concepts such as conditional statements and loops work similarly in different programming languages, even if they do not always look too similar. This is why learning semantics first with simple tools such as visual programming makes it easier to learn how to write code later on as the programmer then only needs to focus on learning the new syntax. [6]

Visual programming tools allow beginners to focus on understanding how programming works without worrying about syntax errors. They can easily explore how concepts such as loops, conditions and variables work by just dragging and connecting blocks. They can build a good solid foundation and understand the semantics before having to worry about the syntax. This way, programming is more about problem-solving than memorising rules and new words. [1]

2.2 Visual programming

Visual programming is a way to program that requires minimal or no writing at all. There are several different types of visual programming such as form-based, node-based and flow-based programming. This thesis will solely focus on block-based visual programming where the user can drag and connect different blocks together and create a code with them. This is the way most popular visual programming sites such as Scratch work.

2.2.1 Scratch, MakeCode and Blockly

Scratch (see Figure 1) is a visual programming language and an online community. It was created by the Lifelong Kindergarten group at MIT Media lab in 2003 and launched as a desktop application in 2007. Scratch is currently developed and maintained by the Scratch Foundation which is a non-profit organisation. [21]

Scratch is the world's largest coding platform for children and it has over 135 million registered users as of May 2025. The target audience is 8 to 16-year-olds although people of all ages are using the platform. [22]

Many schools and coding clubs use Scratch as the first step of learning how to program before moving on to text-based programming languages such as Python. At Suomen Tiedekoulu, Scratch is used for the first half a year of coding lessons which consists of 15 hours of lessons in total. After this, the children move on to program with MakeCode and micro:bits.

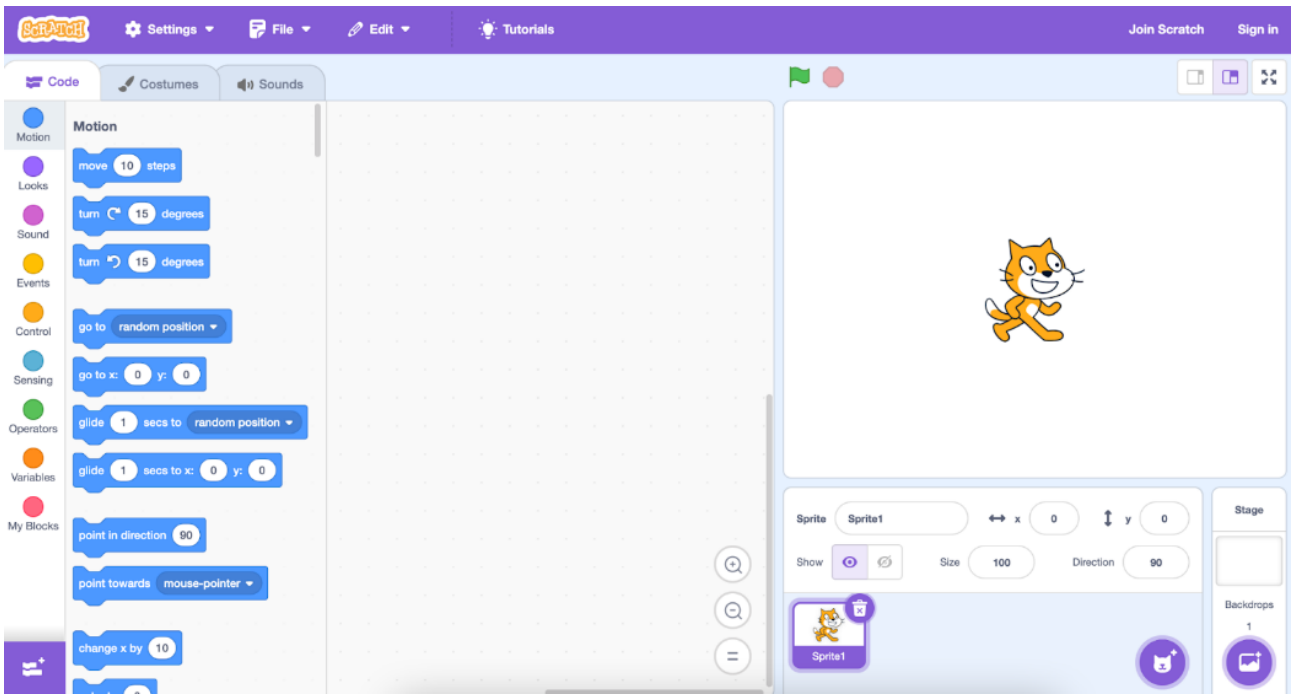

Figure 1: Default Scratch interface showing a new empty project.

MakeCode is a block-based visual programming platform developed by Microsoft. The editor is designed to support a variety of hardware platforms including micro:bit, Adafruit Circuit Playground and others. [23]

Micro:bits were originally developed in 2015 by the BBC as part of their “Make it Digital” initiative. Micro:bits are small versatile computers designed to teach children how to program. They have two buttons, an array of sensors, a 25-LED display and pins for connecting wires and other components as seen in Figure 2. [24]

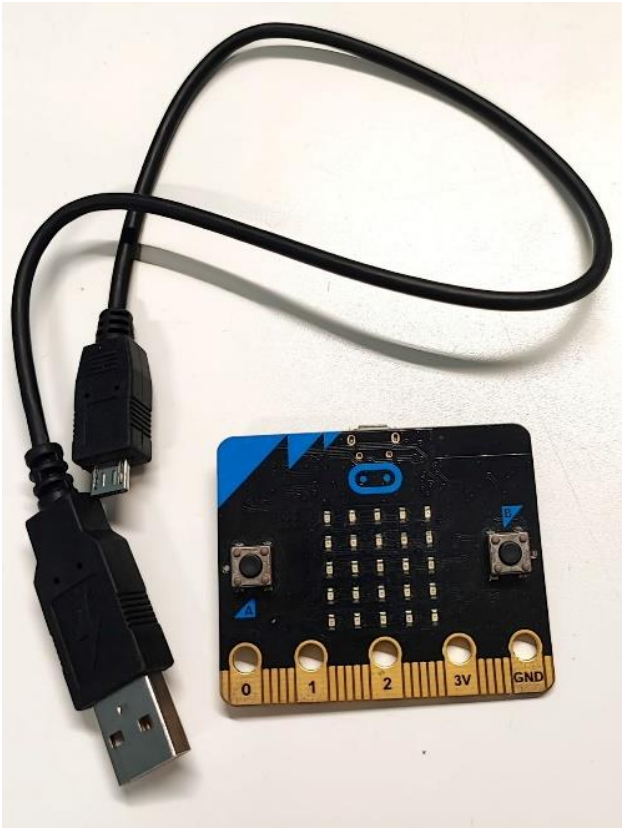

Figure 2: BBC micro:bit with a USB-cable.

With the MakeCode editor, the visual code can either be transferred over to the physical micro:bit or tested quickly with the built-in simulator on the editor (see Figure 3). Having a simulator on the editor allows the user to test their code immediately without needing to upload it to the micro:bit every single time. This instant feedback makes it easier to find mistakes and it also encourages to experiment with different blocks. [23]

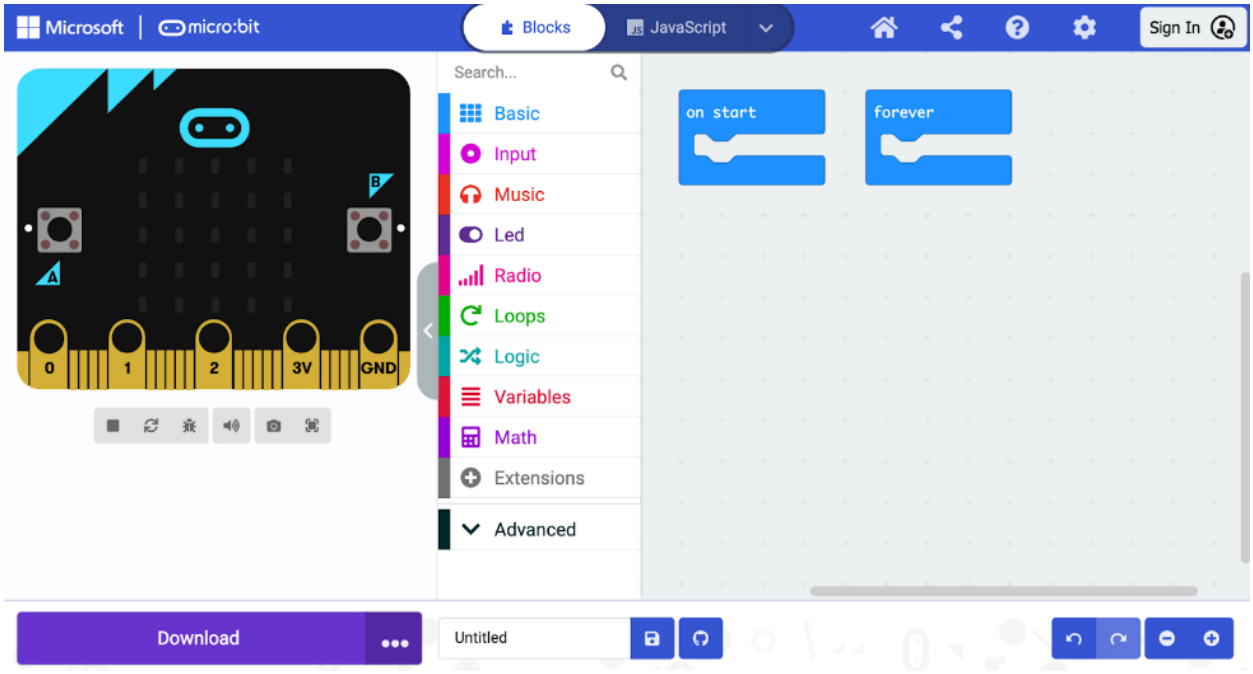

Figure 3: Default MakeCode interface with a micro:bit simulator.

Both MakeCode and Scratch are implemented with the help of the Blockly library. Blockly was first introduced by Google in 2012 and it has become a widely adopted framework for block-based coding. It is a free and open-source web library. [25] A significant moment for Blockly was in 2019, when Scratch 3.0 was launched. This version of Scratch shifted to using the Blockly library. [14]

The Blockly library is the main reason why many block-based visual programming tools resemble each other so much. This can be seen in Figure 4 where the same code has been written with Scratch, MakeCode and Blockly. The top two sets of blocks are made with MakeCode and Scratch. While they both use the Blockly library, they have modified the appearance of the blocks as well as some of the syntax. For example, in the unmodified version of Blockly, numbers are always in separate number blocks. In Scratch and MakeCode, numbers are simply typed into other blocks. Therefore, for example, the code “set variable to 10” is only one block, while in basic Blockly, it is two separate blocks.

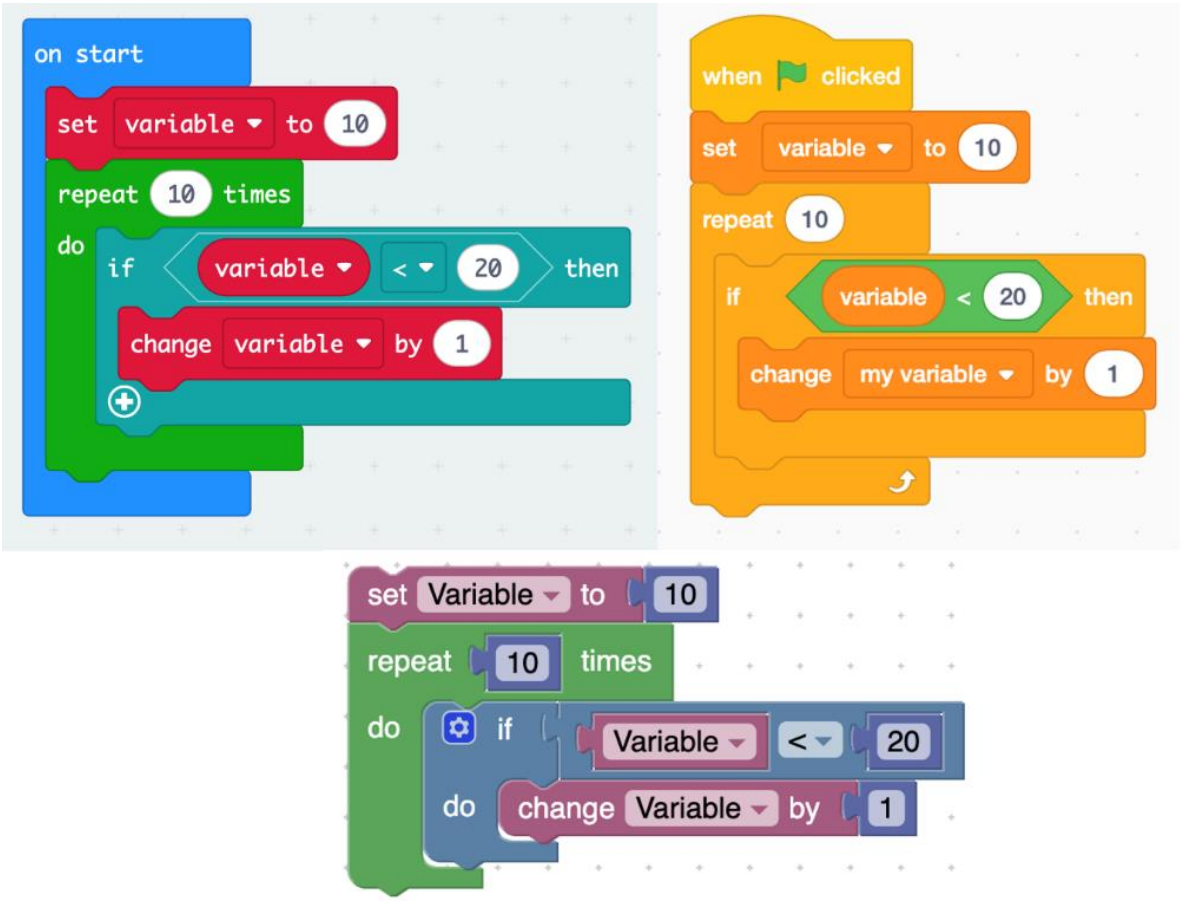

Figure 4: The same code written with MakeCode, Scratch and Blockly.

2.2.2 Syntax

The syntax of visual programming languages is kept as simple as possible to make it accessible for users with little to no experience in programming. This simplicity allows them to learn the basic programming concepts easier and faster. [10]

The code and platforms are often in the programmers own native language, which is not usually an option with text-based programming as they are almost always written in English. With visual programming, non-English speakers can focus more on the logic rather than language barriers. Scratch has also been used as a learning tool to teach English as a foreign language. [1, 5]

In block-based visual programming, the programmer needs to connect different blocks together to create code. This is similar to placing together puzzle pieces. The blocks are often colour coded and divided to different categories which makes it easier and faster to find the

correct blocks. These categories often represent different programming concepts such as loops, conditionals and variables. [10]

Having categories and well organised blocks, leaves more time for thinking about how the code functions and time is not wasted on finding the right commands which can sometimes be the case with text-based programming.

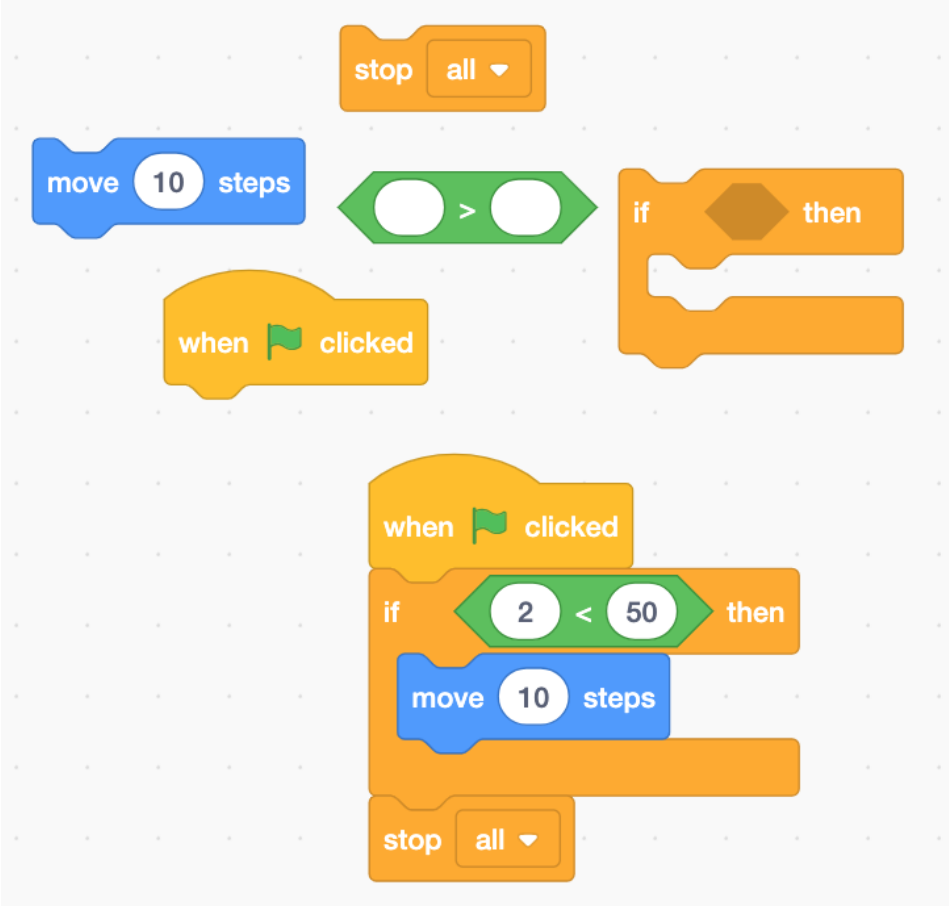

Figure 5: Scratch blocks are designed to only attach to other blocks that they are compatible with.

Blocks can only attach to other blocks that are compatible with them. For example, a block for comparing if something is greater or smaller than something else, will only fit to other blocks where conditional statements are possible as seen on Figure 5. It is easy to tell from the shape of the block that it can only fit together with blocks that share the same shape or have an empty slot with that shape. Blocks that can only be placed at the start or at the end of the program are also shaped differently. This way, the programmer cannot connect together

blocks that are not compatible. It is much harder to get an error compared to traditional text-based coding, where syntax errors are very common. [10]

Because visual programming includes a minimal amount of writing, spelling mistakes do not happen often and they are likely to not affect the way the program works. Having less syntax errors, makes the experience of programming a lot more pleasant especially for younger children who might get frustrated more easily. [10]

The code is read from top to bottom one line at a time. This is the same as with most text-based programming languages. Some visual programming tools, such as ScratchJr, execute code from left to right since this is the way most children are taught how to read. However, it is not common with more advanced visual programming tools and they follow the traditional top to bottom order.

With visual blocks it is also easy to see which code belongs inside a conditional statement or a loop, for example. The blocks are visually nested which makes it clearer when or how many times they will be executed. These blocks are often coloured differently too. This visual structure helps beginners understand how the code flows and also makes it harder to make mistakes. [10]

2.2.3 Semantics

Visual programming tools focus on the semantics rather than the syntax. This is why they are great for learning the logic behind the code. The blocks are connected in a sequence which shows how the program will proceed step by step. Some blocks are also nested inside other blocks and, for example, a loop block will contain other blocks inside it that will be repeated as part of the loop's operation. By seeing how the blocks fit within each other and which blocks are left outside of the blocks, learners can easily understand how loops, conditions and other logic flows work in the program. [10]

Testing the code is usually fast and the program gives immediate feedback or simulation tools such as the characters that move in Scratch. This allows the users to test if their programs behave as expected every time that they make even a small change to it. They can see the logical instructions being translated into actions and correct them if they do not behave

expectedly. This creates a fast feedback loop which makes experimenting with the logic easy and deepens the understanding of how each block works. [10]

Some visual programming tools utilise parallelism. This means it can run several pieces of code at the same time. In Figure 6, the Scratch code has two sets of blocks that both start running when the green flag icon is clicked. The first set of blocks makes the character spin in a circle continuously while the other set of blocks is continuously checking if the space key is pressed. If the key is pressed, then the colour of the character is changed. Changing the colour does not stop the character from spinning, but happens while it spins. In Scratch, code is always attached to either a character or a background. This means that parallelism also happens when different characters or backgrounds are executing their own codes at the same time. [10]

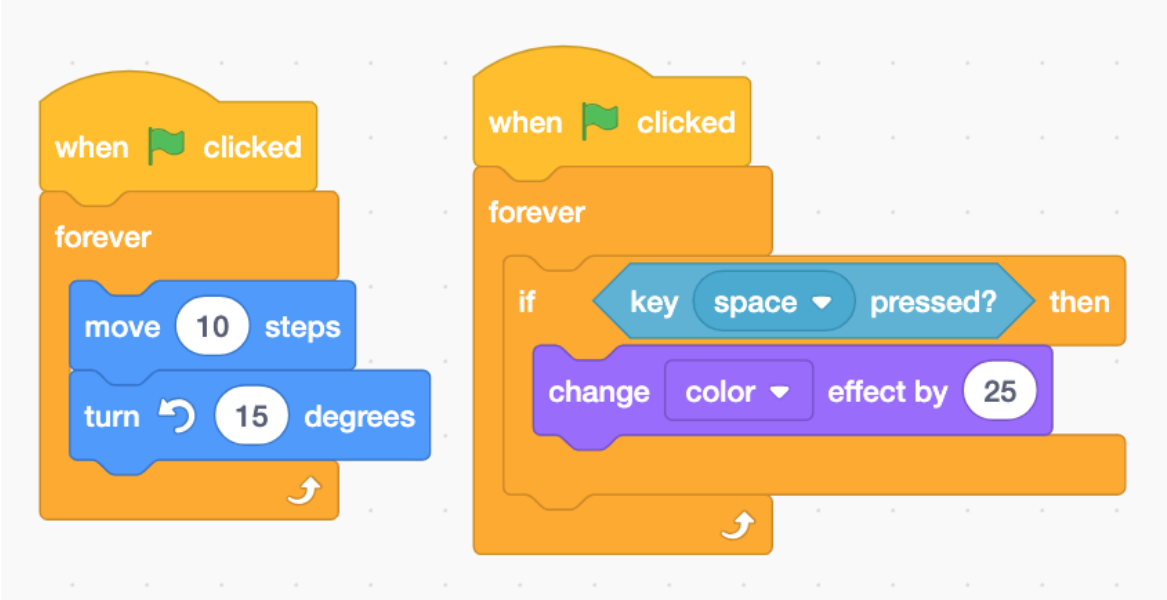

Figure 6: Two or more sets of blocks can be executed at the same time.

Parallelism is an important concept in an advanced programming environment, but it is not easy to implement with basic python knowledge. This can sometimes be confusing for beginners if they are used to using it with visual programming. However, utilising parallelism does teach the concept that modern software often run many tasks at once and the program needs to be designed to handle multiple things at the same time. [10, 12]

In MakeCode for micro:bit, code blocks can be inside “on start” or “forever” blocks. A new empty project always starts with the two blocks already in the workspace as seen in Figure 3. The “on start” block is executed first and then the “forever” block. While there can only be one “on start” block, it is possible to have several “forever” blocks as seen in Figure 7. In this case, the program will execute the first lines of the forever blocks one after each other and then move on to the second lines and execute them one after each other. This happens because the micro:bit runs code sequentially and cannot, for example, show two images or play two sounds at the same time. [26]

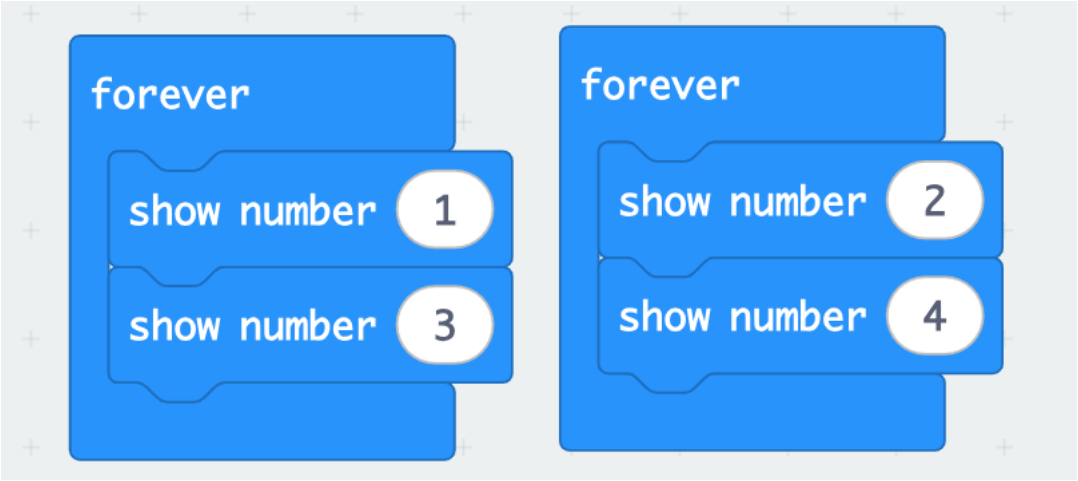

Figure 7: This code will display the numbers 1, 2, 3, and 4 in the correct order when uploaded to a micro:bit.

Visual programming also often emphasises event-driven programming. For example, a micro:bit program usually starts with the “on start” block, while Scratch code can begin with the “when the green flag icon is clicked” block. These triggers initiate the programs. More of these triggers can be seen in Figure 8 and Figure 9. In Scratch, these event blocks run in parallel. On the micro:bit, an event pauses the current code, executes, and then returns back to it. [10, 12, 25]

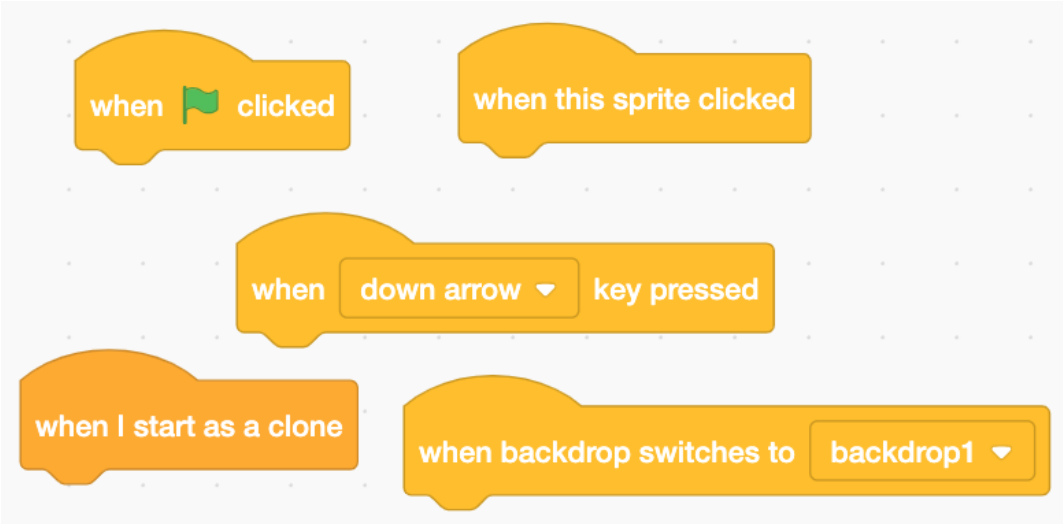

Figure 8: An example of different event blocks in Scratch.

Figure 9: An example of different event blocks in MakeCode.

Event blocks introduce an important semantic concept. Programs can behave differently based on the inputs or events that they receive. The user also learns how software responds to interactions.

In Scratch, when a new variable is created, it automatically appears on the stage and shows what the value of the variable is (see Figure 10). This changes when the variable's value changes in real time, allowing the user to understand variables better and follow along when the program is being executed. This can be toggled off if the user does not want the variable

to be visible on the screen. Variables can store strings, numbers or Boolean values. Unlike with more complex languages, variables cannot store any other types of values. This is the case with most visual programming languages to keep them as simple and intuitive as possible. [5]

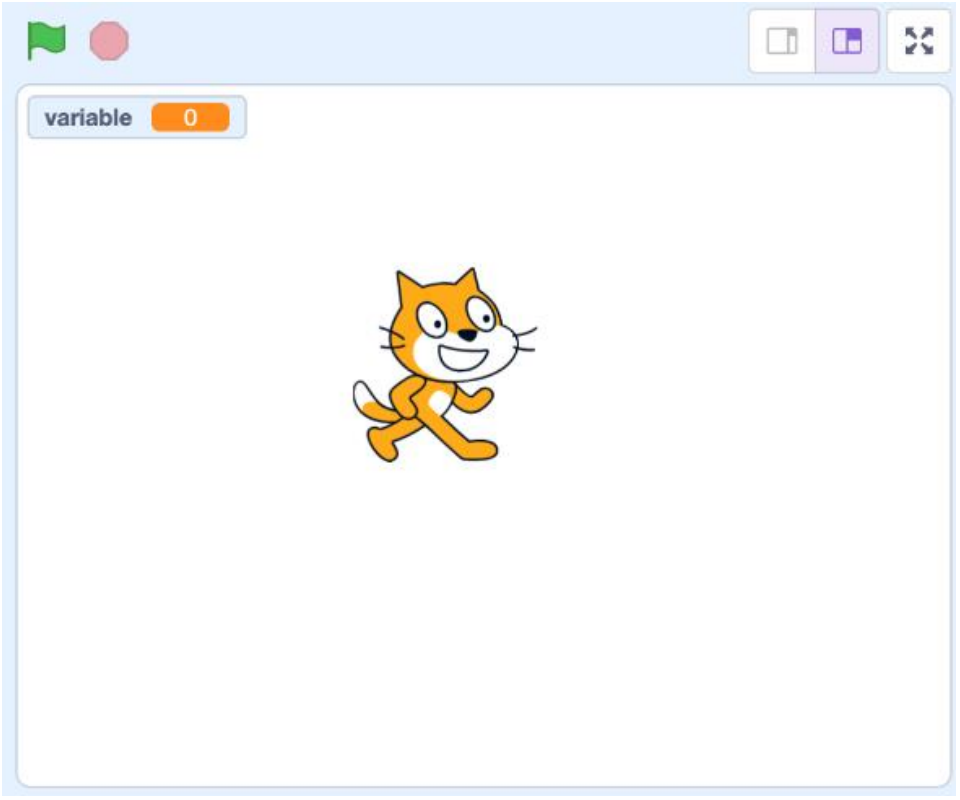

Figure 10: When a variable is created, it appears at the top left corner of the stage in Scratch.

Functions can be made by creating new blocks, as shown in Figure 11. This works similarly to functions in text-based programming languages, where a function is a reusable piece of code that does a specific task. However, MakeCode is the only visual programming tool out of the three examples that calls them functions. [27]

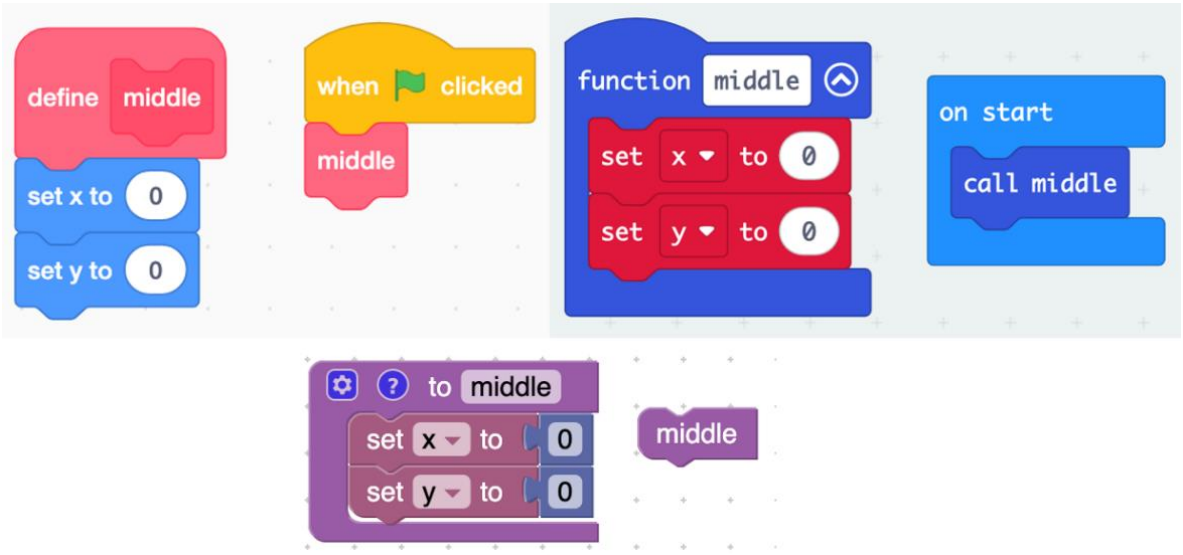

Figure 11: Examples of defining a block called middle that sets x and y variables to 0 shown in Scratch (top left), MakeCode (top right) and Blockly (bottom).

Classes are common in text-based programming languages and they are used to create objects that share common properties and actions. Most common text-based programming languages such as Python, Java and C# are all object-oriented programming languages. [8]

Classes are hard to implement in visual programming and this is why most visual programming tools do not support them. However, Scratch does support clones, which are copies of characters, and they work similarly to objects made from a class. Scratch is an object-based language, but since it does not support inheritance, it cannot be called an object-oriented language. [10]

2.3 Python

Python is a text-based programming language known for its simplicity and readability. This makes it a popular choice for beginners. Unlike visual programming languages that rely on graphical elements and blocks to represent code, Python has a text-based syntax which requires the user to type out the code using words and symbols. [8]

Python was created by a Dutch computer scientist Guido van Rossum and it was first released in 1991. Since then, Python has grown significantly. As of May 2025, Python ranks first in the TIOBE Index which is a monthly ranking of programming languages based on popularity

determined by factors like search engine queries and community activity. This can partly be explained by the increasing popularity of AI and machine learning which heavily uses Python. [28]

2.3.1 Syntax

Python syntax is quite simple and intuitive compared to some other popular programming languages. A 2021 study compared teaching Python and Java, found that Python learners were more motivated and performed better. It suggested that it was easier for beginners because of its shorter syntax and simpler programming structure. [9]

Python uses indentations to define code blocks. In Figure 12, a similar code has been written with Python and Scratch. In Scratch, the blocks are nested inside each other to indicate which parts of the code are inside the loop and condition blocks. In Python, this is done with indentations. Some other programming languages use brackets for this same purpose. Indentations make Python more readable, but an incorrect indentation can lead to syntax and semantic errors. Even one misplaced space can make the code not work at all. This can be very frustrating especially to children. It can also be difficult to understand why "x<50" and "x < 50" are both correct but an extra or missing space at the beginning of the line can cause errors. [4]

```
x = 0
for i in range(10):
    if x < 50:
        x += 6
print(x)
```

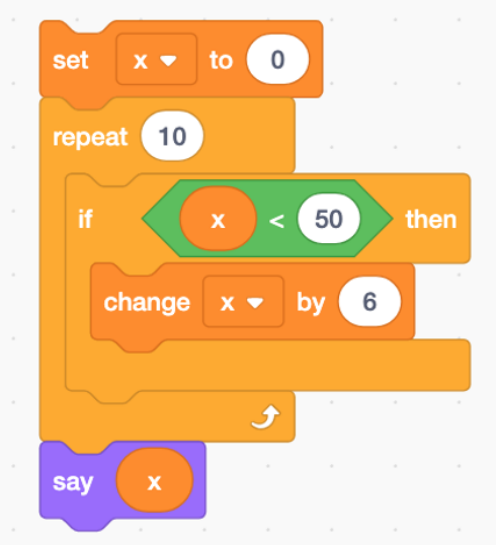

Figure 12: The same code written with Python (left) and Scratch (right).

Unlike in visual programming, it is quite easy to make typos with textual programming. Syntax errors also occur if the user does not understand, for example, differences between "=" and "==". The first one is used when setting values, while the latter is used when comparing values. Capital letters, commas, colons, quotation marks and other special characters can also cause syntax errors easily if they are misplaced or the wrong type. Children might also not be used to typing these characters and it takes time to find them on the keyboard. [1, 4]

2.3.2 Semantics

The basic semantics of Python, other text-based programming languages and visual programming languages are all similar to each other. This is why visual programming is a great tool for learning basic programming concepts before writing text-based code. These include, for example, conditional statements and loops. However, there are several things that work differently in Python compared to visual programming. [8, 12]

For example, to create a loop that repeats ten times as seen in Figure 12, only one block is needed in Scratch. In Python there are several ways of creating this loop, but most often it is created with the line `for i in range(10):`. It can be confusing for children to understand what this line actually does. The variable `i` takes values from 0 to 9, one by one, during each loop cycle. Loops in visual programming tools do not usually include variables. [4]

Creating variables is straightforward with Python. Unlike with some other programming languages, the data type of the variable does not need to be declared when creating the variable. This is the same as with visual programming. However, unlike with most visual programming languages, variables can hold more types of data than just numbers, strings and Boolean values. [10]

Classes are a fundamental concept in Python. A class acts as a blueprint for objects that are instances of that class. For example, a class called `Furniture` can represent different types of furniture. Each object has its own attributes like colours, coordinates and so on. Once a class has been created multiple objects can be generated from it. Each object can be unique while still sharing the same functions. Instead of writing the same code repeatedly for each object, the class has to be only written once and many objects can be created from it and they can all

use the functions in the class. Python also supports inheritance which allows creating new classes that are built on existing ones. [8]

Another important feature of Python and most text-based languages is the use of libraries. Libraries are collections of pre-written code that provide additional functionality. Instead of starting the code from zero, the programmer can import a library and use the functions and classes in it. This saves a significant amount of time and simplifies the coding process. [8]

Visual programming languages like Scratch and MakeCode do not have libraries. This means the concept of libraries must also be learned when transitioning to text-based programming. Parallelism is also not possible in Python without the use of libraries.

3 Creating “Visual Python”

To see if the concept of coding Python visually would be easy to implement and if it can help children with the transition between visual and text-based programming, a website called Visual Python (see Figure 13) was created. On the website users can create Python code with visual blocks. The syntax is kept as similar to actual Python as possible, but the interface is in Finnish which the children are familiar with.

Visual Python was created using the Blockly library. The blocks’ appearance was modified to match with Python’s syntax and the colour’s match with the categories the blocks are in.

Just like in the other visual programming websites, the toolbox is located on the left side and the workspace where users can drag and drop blocks is next to it. On the right side is the print screen that shows what the program prints. The exercises are located under it as well as additional information about Python and Visual Python. The link to the children’s survey is in a tab after the exercises.

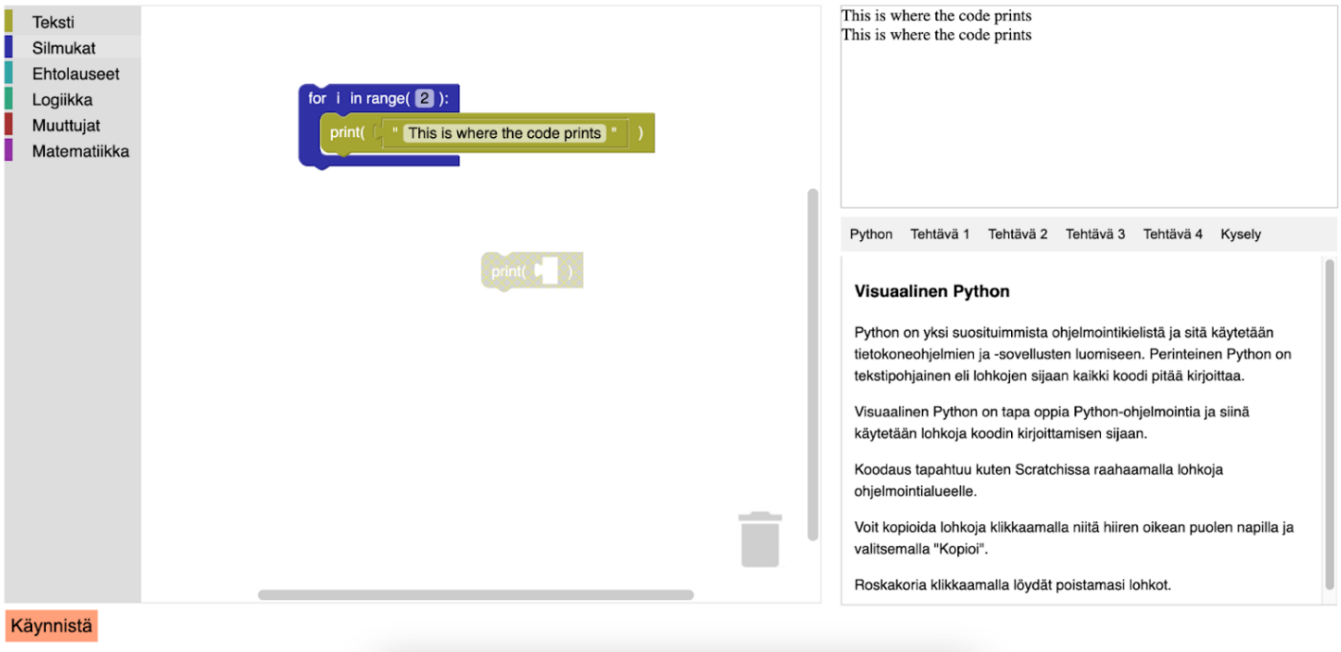

Figure 13: A screenshot of Visual Python.

Visual Python uses the default shape of the Blockly blocks. This is why the appearance differs from the other editors slightly. However, it still familiar enough for children to recognise and

understand how the different blocks interact with each other if they have used any other block based visual programming sites before.

The website does not have as many blocks as other visual programming sites as the purpose of the site is to only test if Visual Python could be beneficial for learning. For example, creating functions or lists is not possible. Visual Python also does not include libraries. However, these could all be added in the future if needed.

The website also does not support parallelism and only one set of blocks can be active at any time. This is why in Figure 13, the single print block is greyed out. Only the longest set of blocks is set active. This is to imitate the behaviour of Python and how it does not support parallelism the same way as most visual programming tools do.

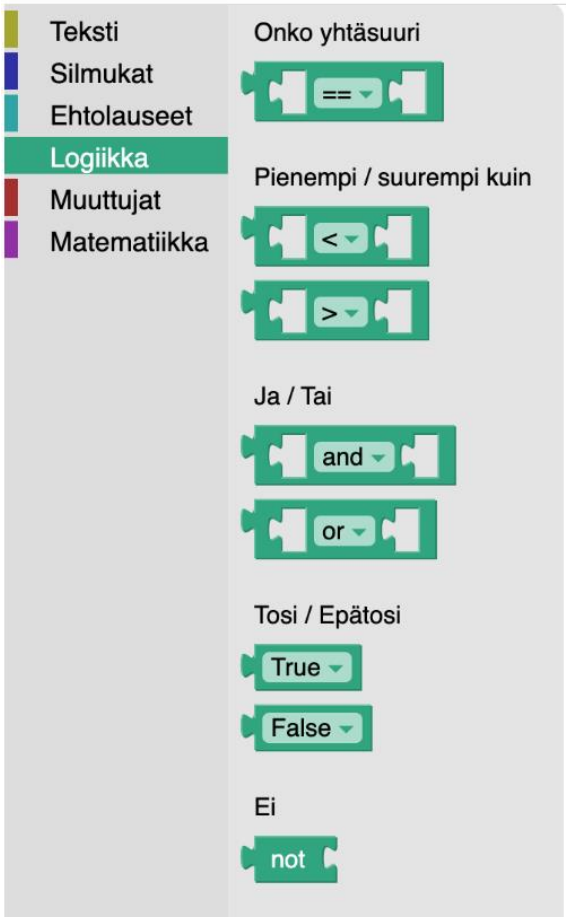

Figure 14: Blocks inside the Logic category.

The blocks are divided into six different categories: text, loops, conditionals, logic, variables and math. Inside these categories, each block has a text above them telling what the words or characters in the block mean in Finnish as seen in Figure 14. This helps the children understand and learn what the new words mean as they might not have learned them in English lessons yet. It also helps them find the blocks they need more easily. The user can also hover over any blocks that have already been dragged to the workspace and the same Finnish text will appear above the block.

4 Evaluation

The evaluation of Visual Python took place in spring 2024, when 82 children around Finland, in both in person and remote lessons, tried out Visual Python for one lesson. Most of the children had never programmed in Python before and they tested the Visual Python website before transitioning to Python programming. The children had four exercises to complete and then a survey to fill out at the end of the lesson.

I personally instructed some of the lessons myself, while the rest were taught by several other coding instructors at Tiedekoulu. Instructors were also asked to complete a survey after they had taught all their lessons. They were asked open questions related to how the children reacted to Visual Python as well as their experiences on the transition from visual programming to textual programming.

4.1 Participants

Most of the children who participated were in 1st to 4th grade as seen in Figure 15. Out of the 82 children who participated in the Visual Python evaluation, only sixteen were girls. This reflects a common trend seen in many coding environments, where fewer girls participate compared to boys.

Twelve of the children were taught remotely while the others were taught in person. There were no major differences in the answers between remote and in person lessons or between girls and boys.

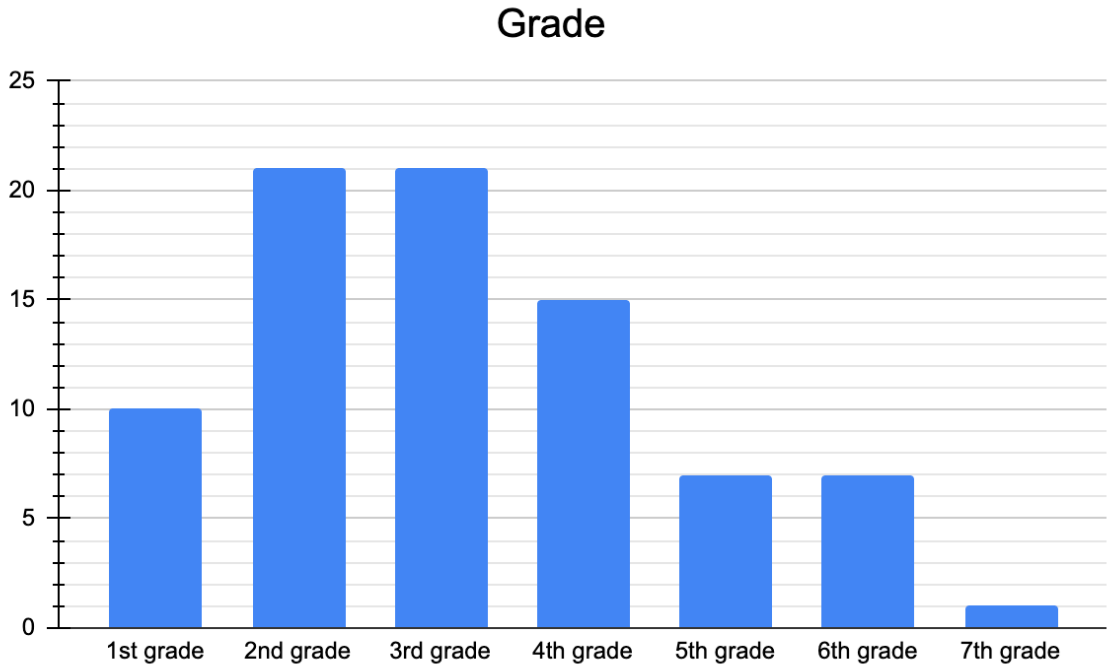

Figure 15: Distribution of all the children across 1st to 7th grade.

Out of all the participants, 26 children had already progressed to Python Turtle exercises or had experience using Python outside of Tiedekoulu. For them, the Visual Python lesson served as a review or opportunity to explore a different way of visualising Python code. These children followed a similar grade distribution as shown in Figure 15. On the other hand, the remaining 56 children were either brand new to Python or had very limited experience, which made the lesson an introduction to Python.

4.2 Lesson structure

Programming lessons at Tiedekoulu last for one hour. This includes setting up the computers and turning them off at the end of the lesson. The instructors were asked to teach one lesson focused on Visual Python and completing the survey, when most of the children in their groups were ready to transition from visual programming to text-based programming.

In this case, the children have started learning programming with Scratch and then moved on to MakeCode with micro:bits and are now ready to start learning Python. However, children

who had already started to learn Python with Python Turtle exercises were also instructed to complete the Visual Python exercises and fill out the survey.

The instructors were told to allocate some time at the end of the lesson so that all the children would have enough time to focus on answering the survey questions. Instructors were also provided with sample answers to the exercises, along with these instructions:

1. At the beginning of the lesson, briefly introduce the Python programming language, similar to how you would normally do when the children start learning Python.

2. Tell the children that we will start by using blocks to write Python code and explore how it is different from coding languages they have already used, such as Scratch and MakeCode.

3. Instruct the children to first read the instructions visible at the bottom right of the website and then move from one tab to the next, starting with exercise 1.

4. The exercises should be completed in order.

5. Guide the children to follow the instructions for each exercise. Only offer help if the instructions are not enough.

6. At the end of the lesson, the children will complete a survey. Everyone must answer the survey, regardless of whether they finished all the exercises or not.

7. Each of the children will answer the survey independently. Responses should not be discussed or shown to friends.

4.3 Python exercises

The children had four exercises to complete during the lesson. The first two exercises included visual instructions with blocks while the other two exercises only featured Python code as references.

4.3.1 Exercises 1 and 2

The first two exercises provided a step-by-step guide for the children to follow, including visual instructions with images showing how the blocks should be connected.

The first exercise introduces the basics of Python with a simple program that asks the user for their name and responds with a greeting. This exercise was designed to help the students understand how variables are made in Python, how user input works and how to output text using the `print()` function. The finished code example can be seen in Figure 16.

The concept of the `input()` function is new to the children as they have not used it in their visual programming exercises. This makes the program more interactive and is used in most of the other exercises as well. The next step of the exercise showed how to use the `print()` function. Printing the variable also helped them see the connection between input, variables and output.

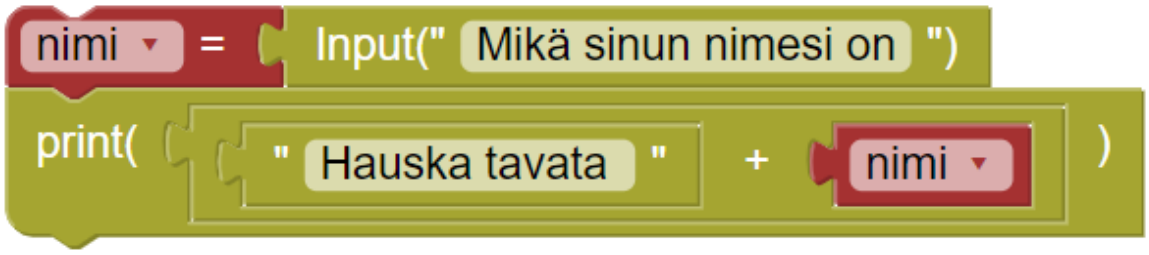

Figure 16: The finished code example on exercise 1.

The second exercise prints a countdown that starts from 10, counts down to 1 and ends by printing “pam!” at the end. The main focus is on the while-loop. The children are already familiar with loops from the “repeat until” or “forever” blocks they have encountered in visual coding platforms, but now they learn the Python syntax for it. The finished code example can be seen in Figure 17.

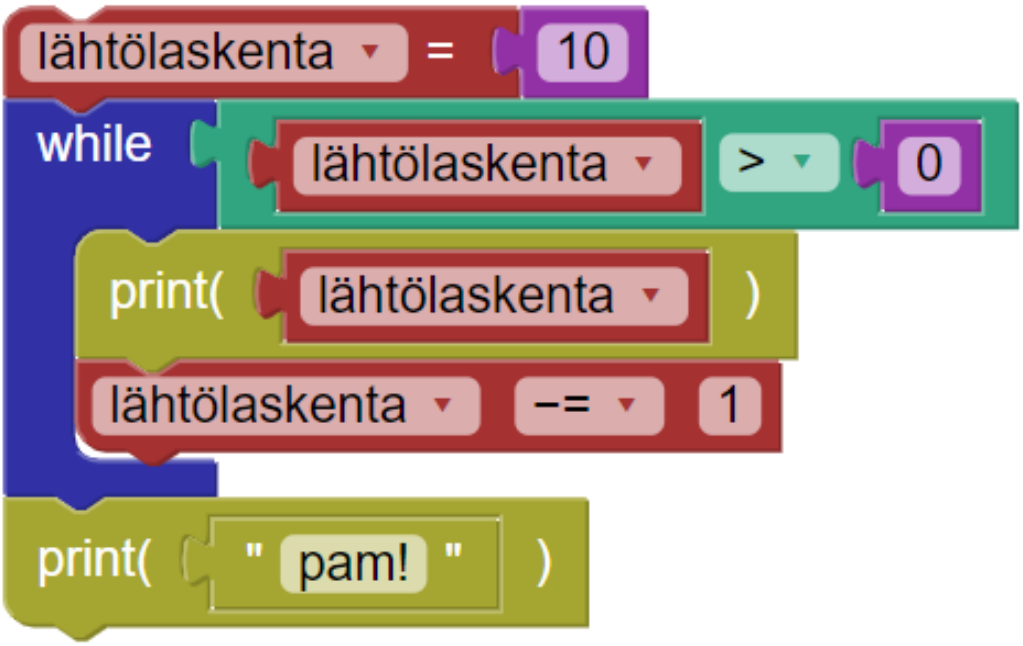

Figure 17: The finished code example on exercise 2.

4.3.2 Exercises 3 and 4

Unlike the first two exercises, the third and fourth exercise do not include any images of the blocks. The code is written out as text, but it is still colour-coded to match the blocks as seen in Figure 18. This is to help the children recognise that the code is still the same whether it is in block form or written as text.

In the third exercise, the program asks the user how old they are and then checks if they are older or younger than ten. This exercise introduces the concept of conditional statements `if`, `elif` and `else` which they have previously used in Scratch and MakeCode. Now for the first time the syntax is in English for them and they learn what the keyword `elif` means.

```
ikä = input("Kuinka vanha olet")
if ikä >= 10:
  print("Olet yli kymmenen")
elif ikä < 10:
  print("Olet alle kymmenen)
else:
  print("Et syöttänyt numeroa")
```

Figure 18: The finished code example on exercise 3 written in Python without any blocks.

The final exercise is a guessing game (see Figure 19). It starts by generating a random number between 1 and 10 using Python's random function. The goal is to get the user to guess the correct number by using a while loop that keeps running until the user guesses correctly. If the guessed number matches the random number, the game congratulates the player and breaks out of the loop which ends the game.

This exercise shows how to create a forever-loop in Python and how to end it. If the children have finished the exercises early, there is an extra challenge where the children are given the option to limit the number of guesses by using a for-loop instead of a while-loop.

```
numero = random.randint(1,10)
while True:
  arvaus = input("arvaa numero")
  if arvaus == numero:
    print("numero oli: " + numero)
    break
  else:
    print("arvasit väärin")
```

Figure 19: The finished code example on exercise 4.

4.4 Survey results

After completing the exercises, the children answered a survey at the end of the lesson. The survey asked for their background information as well as seven multichoice questions. The focus here is on the answers from the students who had not done any Python programming before.

The other group of children, who had experience with Python programming already, also filled out the same survey and their results are briefly presented at the end of this subsection, but they are not included in the bar diagrams.

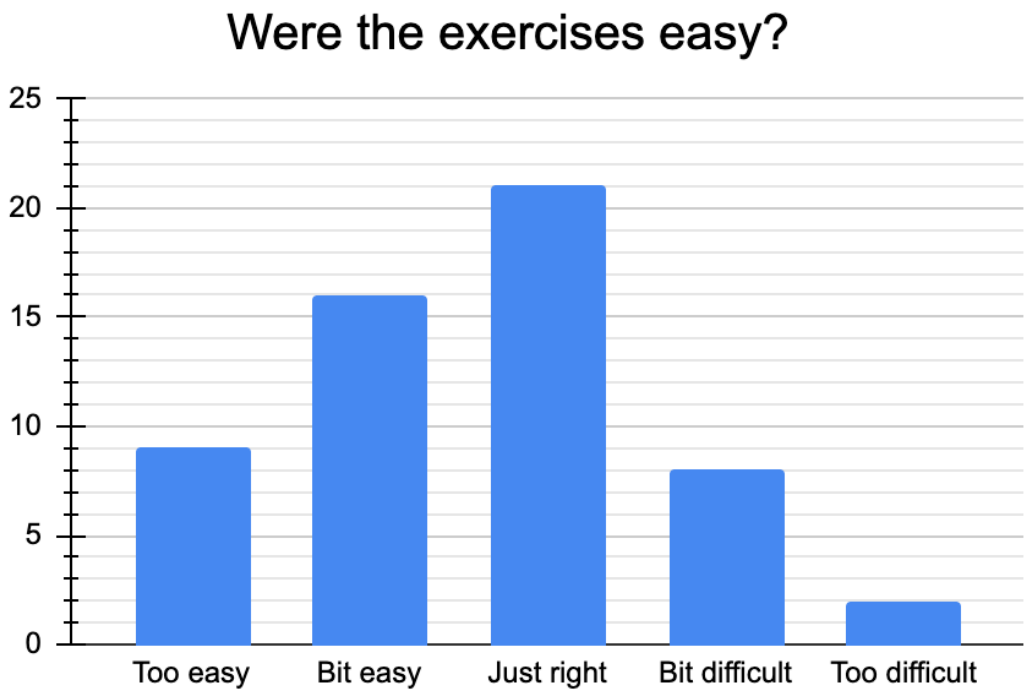

Figure 20: Responses to the question “Were the exercises easy?” from children without previous Python experience.

The first multichoice question asked if the children thought that the exercises were easy. Most of the children felt that the exercises were the right level for them as seen in Figure 20. The biggest group said they were “just right” followed by “bit easy”. Only two children said they were “too difficult”. This suggests that the exercises were well balanced for the group.

When it came to understanding what the different blocks do, most of the children said they understood most or all of them as seen in Figure 21. Only a few said they did not understand what any blocks do. However, some of the children still said they only understood some or half of the blocks, so not everyone felt fully confident or knew what they do.

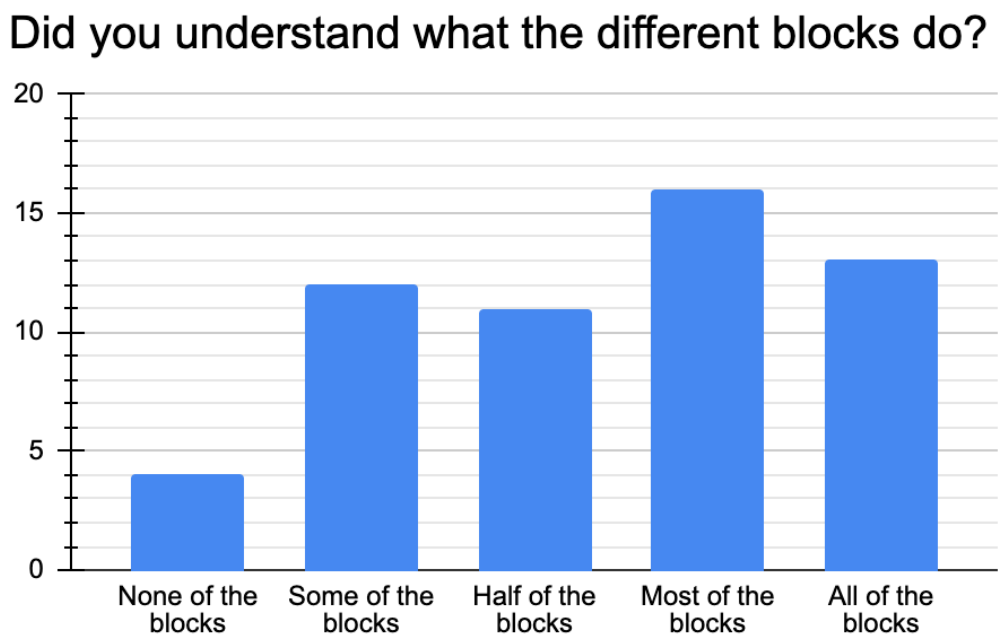

Figure 21: Responses to the question “Did you understand what the different blocks do?” from children without previous Python experience.

The children were also asked if they knew what the words “if”, “else”, “while”, “True”, “False”, “break” and “print” mean in English (see Figure 22). The children have only programmed in Finnish before and some have not started taking English lessons yet. Most of the children said that they knew at least a few of the words and some said they knew all of them. But the biggest group said they did not know any. This means that despite the fact that many of the children did not know what the keywords meant in English, they felt confident that they knew what the blocks do when doing the exercises.

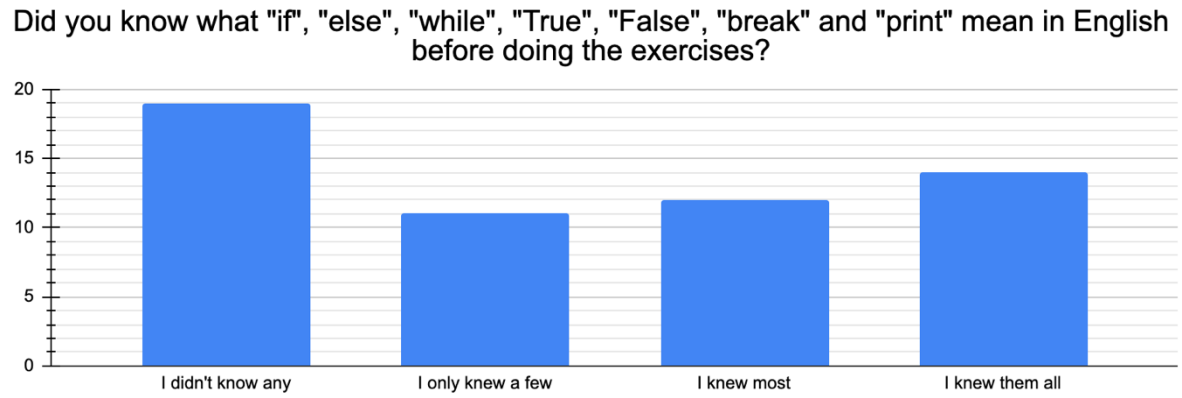

Figure 22: Responses to the question “Did you know what “if”, “else”, “while”, “True”, “False”, “break” and “print” mean in English before doing the exercises?” from children without previous Python experience.

Most children completed all of the four exercises as seen in Figure 23. Out of the 56 children without previous Python experience, ten children stopped after two or three exercises and only two did just the first exercise. This shows good engagement and that the children were able to keep doing the exercises even when the instructions switched from using blocks to standard Python code.

Figure 23: Responses to the question “Did you finish all the exercises?” from children without previous Python experience.

The children who completed at least the first three exercises were then asked if exercises three and four were harder than the first two (see Figure 24). Most of the children said they were "moderately more difficult" followed by "slightly more difficult" and "equally difficult". Out of these 46 children, twelve found them "much more" or "significantly more" difficult. So, although the later exercises were a bit harder, most children did not feel that they were too challenging. This shows that the children acknowledged the difference, but did not find it too hard to follow instructions written in Python code instead of being represented by blocks.

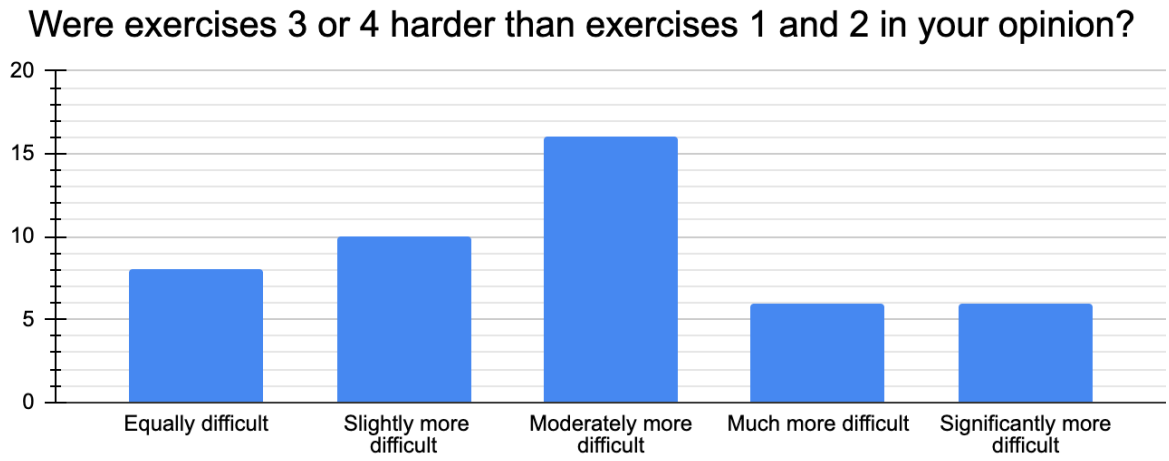

Figure 24: Responses to the question "Were exercises 3 or 4 harder than exercises 1 and 2 in your opinion?" from children without previous Python experience and who completed at least three exercises.

The last two questions of the survey showed the children a short code and asked them what it would print. The first code was done with Visual Python blocks as seen in Figure 25. There are two options for what the code might print and if the children did not know the answer, they were encouraged to answer "I don't know" instead of guessing. This was to help understand what they actually knew and to avoid lucky guesses.

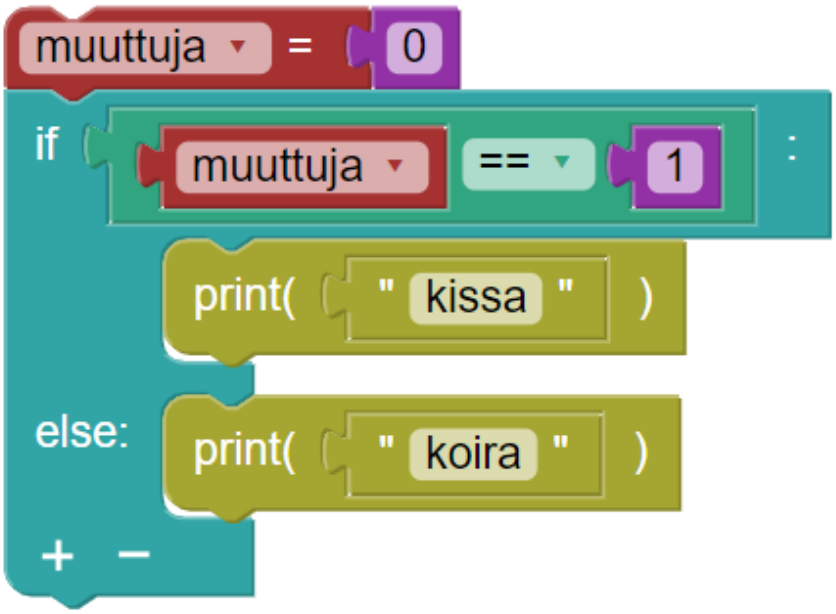

Figure 25: Visual Python code in code question 1.

The second code question used regular Python code instead of blocks as seen in Figure 26. The structure and logic were still simple and similar to the previous question, but this time the children had to read plain text code and think what it would print.

```
muuttuja = 5
if muuttuja >6:
    print("porkkana")
else:
    print("peruna")
```

Figure 26: Python code in code question 2.

Half of the children answered “I don’t know” to both of the questions. The first question was answered correctly by 35.7% of the children (see Figure 27) and the second question was answered correctly by 32.1% (see Figure 28).

This small drop of 3.6 percentage points suggests that the blocks might have helped a few students follow the logic more clearly, but overall, the difference was not very large. In both cases, most children either were not sure or got the answer wrong. This shows that many of them still found it difficult to read and understand what a short program would do, even when the code looked similar to what they had just worked with. This suggests that the children did not actually understand what all the blocks do even if they did feel confident that they did.

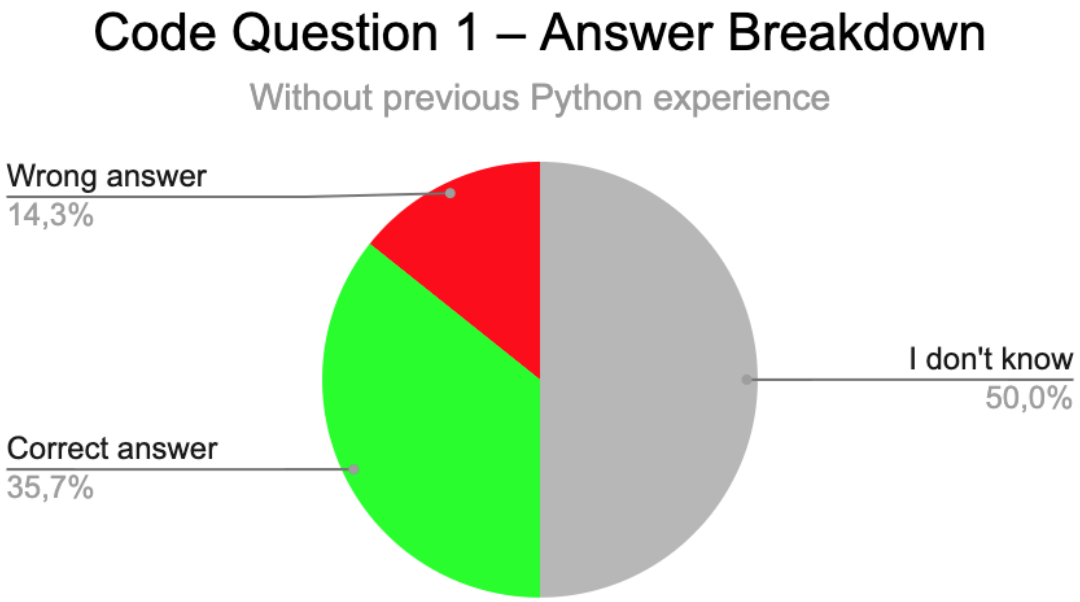

Figure 27: Answer breakdown of the first code question from children without previous Python experience.

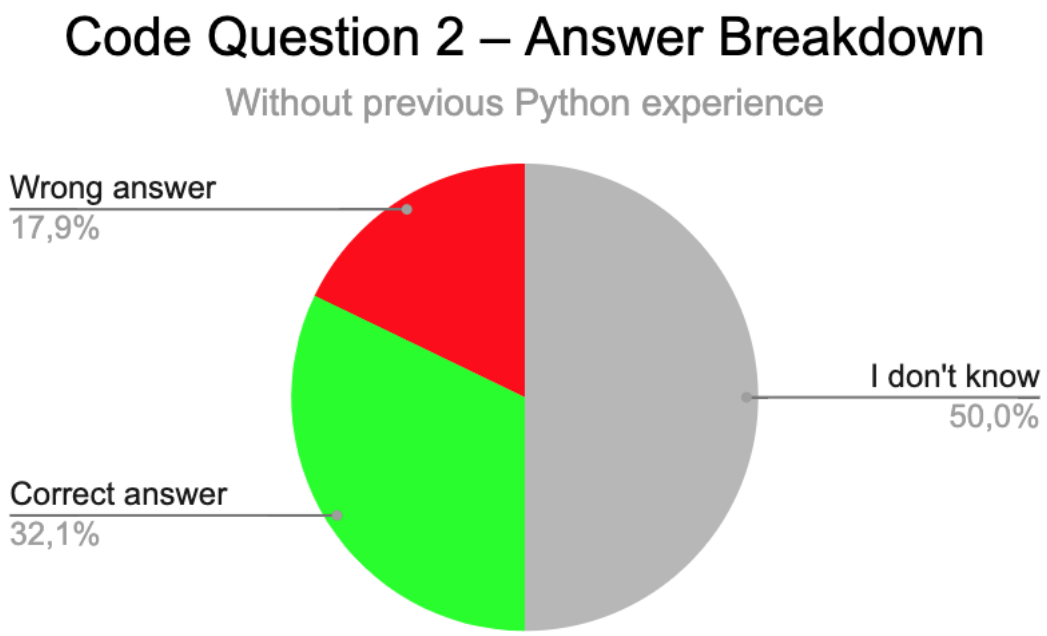

Figure 28: Answer breakdown of the second code question from children without previous Python experience.

When comparing these results to the 26 children who had done Python exercises before, it is clear that the children with previous Python experience found the exercises easier and got better results on the code questions.

Most of these children said the exercises were “bit easy” or “just right” and no one said they were too difficult. Almost all of them said they understood most or all of the blocks. This group also finished more of the exercises compared to the others. Most of them completed all four.

In the final code questions, this group performed clearly better than the other group. Only 15.4% answered “I don’t know” to either question. This shows that these children felt more confident in their answers. The first code question with the Visual Python blocks, was answered correctly by 46.2% (see Figure 29), compared to 35.7% by children without any previous Python experience. However, more children also answered the questions incorrectly.

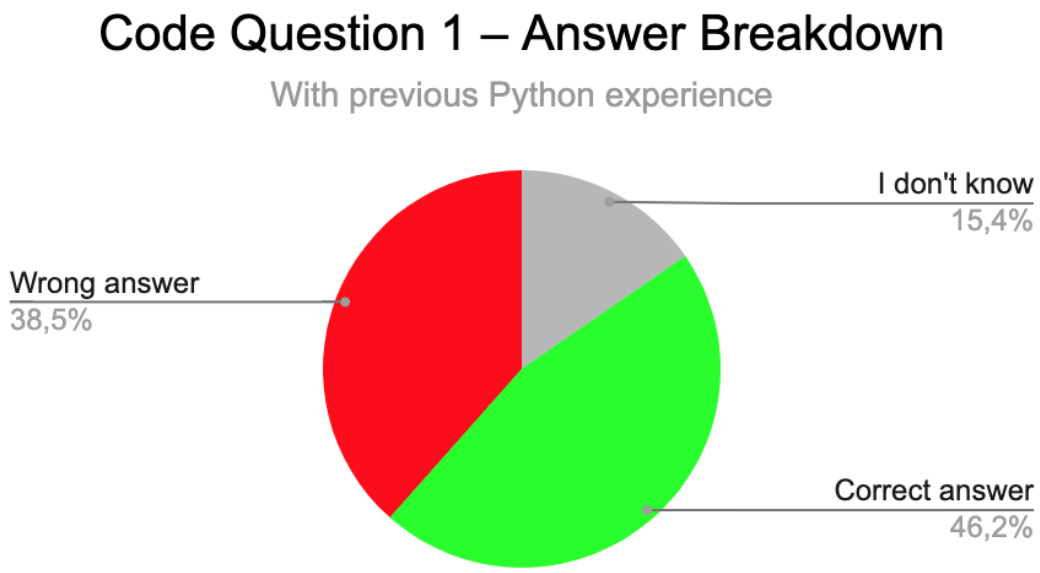

Figure 29: Answer breakdown of the first code question from children with previous Python experience.

The second question with plain Python code was answered correctly by 50% of the children with previous Python experience (see Figure 30), compared to 32.1% of children without previous Python experience. This is a difference of 17.9 percentage points.

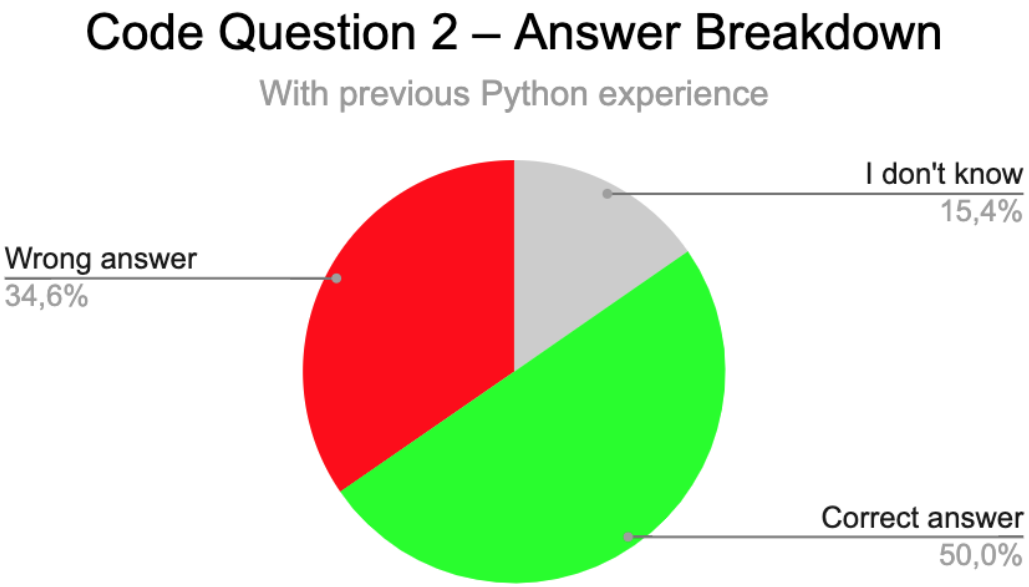

Figure 30: Answer breakdown of the second code question from children with previous Python experience.

4.5 Instructors' survey

The instructors were told to fill out a survey once all of their groups had finished with the Visual Python lessons and filled out their own surveys. Seven of the instructors answered the survey. They were asked which city they teach in and how long they have worked as coding instructors as well as five open-ended questions. All of the instructors had worked as coding instructors at least for half a year or more.

The first open-ended questions asked about the children's reactions to Visual Python and whether the instructor thought that the exercises were too easy or too difficult for their groups. Most of the instructors expressed that the children enjoyed the exercises and some had even enjoyed them as much or more than the micro:bit exercises they had been doing on previous lessons.

> *Instructor 2*: Everyone in the group said they liked it more than, for example, micro:bit. This was also visible in their increased enthusiasm.
>
> *Instructor 5*: They (the children) were interested and apparently felt it was just as easy as using micro:bit blocks in MakeCode.

The instructors noticed a difference between exercises one and two compared to three and four, and said that the children had more difficulties with the last exercises.

> *Instructor 4*: The first two exercises were easy, but the last two were more difficult. With assistance, most of the children were able to complete at least 3 exercises.
>
> *Instructor 7*: Some (children) seemed to enjoy the challenges up until exercise 3.

Most instructors said the level of difficulty was good for the majority of children in their groups, but extra tasks for the faster students would have been helpful. They also expressed that the children can be fast to follow instructions, but might not understand the actual logic behind the code. This also seems to be the case when looking at the children's answers to the code questions.

> *Instructor 1*: For some (children), the exercises were very easy, while others struggled to find the correct blocks or words. The comparison signs were still difficult for some.
>
> *Instructor 2*: All the students did well and most completed the exercises reasonably early (several about 15-20 minutes ahead of time). They only needed a

> little guidance. In general, the level was suitable, but perhaps a bit too easy, or could have used one more challenging additional task for the faster ones.
>
> *Instructor 6:* They (the exercises) seemed good for children who have actually understood how variables work.
>
> *Instructor 7*: The last exercise was too difficult. Those who managed to get started could connect the blocks and Python text if the text matched exactly with what was in the blocks. The difficulty is in reading comprehension. Most can copy the example code blocks, but they don't yet understand what they are reading in the exercise instructions. The exercise instructions seemed to cause frustration because they couldn't understand them on their own.

The instructors acknowledged that some children have struggles with the transition between visual programming and textual programming and one instructor was already teaching Python gradually with the help of MakeCode's code section where it can turn code created with blocks to Python or JavaScript code.

> *Instructor 3*: For some children, it is challenging to apply what they have learned to Python, but some understand it right away.
>
> *Instructor 5*: The next programming course I have instructed in a way that the code is first created using blocks, but then explained through the MakeCode's code section, showing how the blocks affect the code. I've also explained the code section using Python and JavaScript, without requiring the students to write it themselves yet.
>
> *Instructor* 7: Yes, I have noticed the difficulty. I have young children in my group who are not yet reading properly and cannot type long sentences on a keyboard. Copying code is uncreative work for them and slow when transitioning to typing. This causes frustration for the young ones.

The second to last question focused on the instructors' own opinions of Visual Python and whether they felt it was useful for the children. All instructors agreed that it would be beneficial at least to some degree.

> *Instructor 2*: Although it is very similar to Scratch or the micro:bit platform, the students found it different and more fun. I believe it will help in transitioning to text-based programming, at least to some extent.
>
> *Instructor 3*: It will definitely be helpful for at least some of the children. It makes understanding the logic of coding easier for others.

> *Instructor 4*: I believe Visual Python is helpful when transitioning to written Python, as it presents the words in a more visual form.

The last question of the survey gave the instructors a chance to share anything else they might want to say or give feedback about. Instructor 5 suggested an approach where the children could see visual blocks and text-based code dynamically at the same time and edit them both as they wish. This approach is similar to the BrickLayer mentioned earlier created by Cheung et al. [3].

5 Conclusion

The final chapter addresses the three research questions that were presented in the first chapter. The chapter goes over each of the research questions one at a time and the findings from previous chapters are presented. This chapter also discusses potential future work and topics of interest.

5.1 Answers to the research questions

Each of the three previously established research questions are in their own subsections where the main results of this thesis are presented.

5.1.1 Research question 1

RQ1: Do children find it difficult to move from visual programming, using drag-and-drop blocks, to typing out code in languages like Python?

The instructors who participated in testing Visual Python expressed that they think the transition is difficult for some students especially if they still have problems with reading comprehension or understanding how variables work. It is difficult for some children to apply what they have learned with visual programming to Python. One instructor also found it better to demonstrate how Python works with MakeCode's code section, where it can turn code created with blocks to Python or JavaScript code, before the children have to write code themselves.

Other papers written about the subject have also observed difficulty with the transitioning. Cheung et al. [3] found that there is a gap between older and younger students who find graphics-based environments too limited and textual environments too difficult. Kaurel [7] has also experienced this first hand and noticed students ending their coding hobbies because of the struggle.

Based on the children's survey results, it seems like children are good at following instructions and confident in their own abilities, but they might not fully understand the logic

behind code. This is an aspect which importance becomes more significant in textual programming.

5.1.2 Research question 2

RQ2: Can the transition be made easier by breaking it down and adding a step between visual and textual programming? Showing Python in a block format before moving fully to text might make it easier for children to understand Python syntax before having to actually write it themselves. Does this method actually help children learn Python better before they start writing it themselves?

All instructors found Visual Python helpful at least in some ways and one suggested an approach where children code visually but a text-based code appears dynamically. Other research on the topic have also explored this idea as well as suggested other options. They all came to the conclusion that adding some kind of extra step between visual and textual programming can help children with the transition. However, there are several ways of easing the transition and it is likely to depend on the learner which one would be the best for them. No research was found that compares different approaches.

The children, who participated in testing Visual Python, found Visual Python easy to use and said the exercises were mostly easy. Most of the children did not know the English keywords needed in Python, but said that after the exercises they understood what most of the blocks do. It is likely that learning Python would be easier if children have easy access to translations of Python keywords in their own native language, regardless of the way they transition to textual programming.

The children also thought that instructions given in Python code instead of blocks were moderately more difficult, but not too difficult. However, their answers to the code questions show that the participants did not understand simple if else statements. It is possible that it would be easier to teach children how to read Python code before they write it. This way they would not have frustration with syntax errors.

5.1.3 Research question 3

RQ3: If children use a visual version of Python, will it help them understand the actual text-based Python code better before they ever type it themselves?

Based on the answers to the code questions, it seems like children who understood the question presented with Visual Python code were almost as likely to answer correctly when a similar question was written in regular text-based Python. This suggests that using Visual Python may help children grasp the concept and syntax of Python before they start typing it themselves.

5.2 Future work

There are several ways to ease the transition from visual to textual programming and Visual Python is just one approach to this challenge. Another approach is generating text-based code with visual blocks in real time. This seems to be a popular way to demonstrate how visual programming concepts apply in textual programming.

Kaurel [7] explored the idea of adding a new simple programming language for the transition that would first be programmed visually and then textually. However, adding another language could possibly confuse the learners rather than help them. This is why an approach like Visual Python might be more beneficial. The idea of generating text-based code with visual blocks could be applied to Visual Python as well and it would be an interesting subject to explore further. Jumping straight from visual to textual programming without any examples showing how visual programming concepts translate to textual programming certainly does not seem to be the easiest way for most learners in general.

It is possible that it would be easier to teach children how to read Python code before they start writing it. This way they would not experience frustration with syntax errors while still getting familiar with the language. However, it could be said that it is impossible to gain deeper understanding for anything, if you never try doing it yourself. Just as playing a piano cannot be learned just by reading a book about it. That said, it could still be useful for the first steps of learning.

It would be interesting to study this transition further and have comparison groups where children follow these different learning approaches. However, organizing such groups can be

difficult especially outside of school settings since children often join or leave hobby groups irregularly. So far, it seems like no research has been done comparing these different approaches.

References

1. Armoni M, Meerbaum-Salant O, Ben-Ari M (2015) From Scratch to “Real” Programming. *ACM Transactions on Computing Education* 14:1–15. doi: 10.1145/2677087

2. Bati K (2022) A systematic literature review regarding computational thinking and programming in early childhood education. *Education and Information Technologies* 27:2059–2082. doi: 10.1007/s10639-021-10700-2

3. Cheung JCY, Ngai G, Chan SCF, Lau WWY (2009) Filling the gap in programming instruction: a text-enhanced graphical programming environment for junior high students. *ACM SIGCSE Bulletin* 41:276–280. doi: 10.1145/1539024.1508968

4. Church K, Rolon-Mérette T, Ross M, Rolon-Mérette D (2021) Introduction to Python’s Syntax. *The Quantitative Methods for Psychology* 17:S1–S12. doi: 10.20982/tqmp.17.1.S001

5. Costa S, Gomes A, Pessoa T (2016) Using Scratch to Teach and Learn English as a Foreign Language in Elementary School. *International Journal of Education and Learning Systems* 01

6. Hardin T, Jaume M, Pessaux F, Donzeau-Gouge V (2021) *Concepts and semantics of programming languages: a semantical approach with OCaml and Python* ; 1. ISTE, Ltd, London

7. Kaurel HG (2016) Easing the Transition from Visual to Textual Programming. MSc Thesis, Norwegian University of Science and Technology, Trondheim, Norway

8. Lee KD (2017) *Foundations of Programming Languages*. Springer International Publishing, Cham

9. Ling H-C, Hsiao K-L, Hsu W-C (2021) Can students’ computer programming learning motivation and effectiveness be enhanced by learning python language? A multi-group analysis. *Frontiers in Psychology* 11:600814. doi: 10.3389/fpsyg.2020.600814

10. Maloney J, Resnick M, Rusk N, Silverman B, Eastmond E (2010) The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* 10:1–15. doi: 10.1145/1868358.1868363

11. Mancino D (2023) Digital literacy in the EU: An overview. https://data.europa.eu/en/publications/datastories/digital-literacy-eu-overview. Accessed 23 Feb 2025

12. Meerbaum-Salant O, Armoni M, Ben-Ari M (Moti) (2013) Learning computer science concepts with Scratch. *Computer Science Education* 23:239–264. doi: 10.1080/08993408.2013.832022

13. Nojonen T (2024) Ohjelmoinnin opetus lukioiden paikallisissa opetussuunnitelmissa. MSc Thesis, University of Turku, Finland.

14. Pasternak E Scratch 3.0’s new programming blocks, built on Blockly. In: Google Developers Blog. https://developers.googleblog.com/en/scratch-30s-new-programming-blocks-built-on-blockly/. Accessed 26 Feb 2025

15. Quander K, Milky TR, Aponte N, Caceres Carrascal N, Woodward J (2024) “Are you smart?”: Children’s Understanding of “Smart” Technologies. In: *Proceedings of the 23rd Annual ACM Interaction Design and Children Conference*. ACM, Delft Netherlands, pp 625–638

16. Romero M (2024) Lifelong learning challenges in the era of artificial intelligence: a computational thinking perspective. *International Research Meeting in Business and Management* doi: 10.48550/ARXIV.2405.19837

17. Sáez-López J-M, Román-González M, Vázquez-Cano E (2016) Visual programming languages integrated across the curriculum in elementary school: A two year case study using “Scratch” in five schools. *Computers & Education* 97:129–141. doi: 10.1016/j.compedu.2016.03.003

18. Smahel D, Machackova H, Mascheroni G, Dedkova L, Staksrud E, Ólafsson K, Livingstone S, Hasebrink U (2020) EU Kids Online 2020: Survey results from 19 countries

19. Tanhua-Piiroinen E, Kaarakainen S-S, Kaarakainen M-T, Viteli J (2020) Digiajan peruskoulu II. https://julkaisut.valtioneuvosto.fi/handle/10024/162236. Accessed 24 Feb 2025

20. Tóth T, Lovászová G (2021) Mediation of Knowledge Transfer in the Transition from Visual to Textual Programming. *Informatics in Education* 20:489–511. doi: 10.15388/infedu.2021.20

21. Our Story. In: Scratch Foundation. https://www.scratchfoundation.org/our-story. Accessed 25 Feb 2025

22. Scratch statistics. https://scratch.mit.edu/statistics/. Accessed 22 May 2025

23. About. In: Microsoft MakeCode. https://makecode.com/about. Accessed 26 Feb 2025

24. About. In: The Micro:bit Educational Foundation. https://microbit.org/about/. Accessed 26 Feb 2025

25. Blockly. In: Google for Developers. https://developers.google.com/blockly. Accessed 26 Feb 2025

26. The micro:bit - a reactive system. In: Microsoft MakeCode. https://makecode.microbit.org/device/reactive. Accessed 12 Mar 2025

27. Function definition. In: Microsoft MakeCode. https://makecode.microbit.org/types/function/define. Accessed 12 Mar 2025

28. TIOBE Index. In: TIOBE. https://www.tiobe.com/tiobe-index/. Accessed 22 May 2025